A Performance Model for Parallel Programs

Abstract

In this paper, we describe a model for determining the optimal data and computation decomposition for a parallel program by predicting its execution time. The model takes into account various types of data and computation decompositions for each loop nest and combines these to determine a global optimum. The unique features of the model are its accuracy, platform independence, and ability to take potential dynamic decompositions and interleaving of computation and communication into account. We give performance results for the application of the model to standard benchmarks on the IBM SP/2 and a Network of Workstations.

1 Introduction

Parallel computers are used extensively to solve compute intensive problems such as weather prediction, automobile crash simulation and medical imaging. Applications are usually implemented using platform independent standards, such as MPI and HPF. However, substantial programmer time must be invested in fine-tuning the performance of a parallel application on each target computer. The performance of an application depends on the mapping (decomposition) of computation and data to the nodes of a parallel computer. We have developed a compile-time analysis tool, which given a sequential dense-matrix application[1], profiling information about sequential loop execution times, and certain target machine parameters, determines the optimal parallelization for the target platform. This parallelization is specified by

[1] A dense matrix application is one in which most array indices are affine expressions, e.g. `A[I+3, 2J]`

generating a globally optimal mapping of both data and computation to processors for each loop nest, which is then implemented using portable MPI calls. Even if the performance estimators for individual loop nests are 100% accurate, determining the global optimum is NP complete. Hence, we use a greedy heuristic to search for a global optimum rather than an exhaustive search. In practice, the greedy heuristic performs well.

Our toolkit utilizes SUIF[1] to statically analyze a sequential program to determine available parallelism in loop nests. Sequential execution times and frequencies of execution of loops are determined by profiling, although they could also be estimated by compile-time analysis. Performance prediction is performed by combining an analytical model described below together with a few simple machine parameters (such as a linear cost model for communication).

The organization of this paper is as follows. Firstly, we describe the PAL graph, a program representation that summarizes the data and computation flow and available parallelism. Then we describe the performance model used to estimate the execution time of loop nests including communication times. Finally, we discuss the accuracy of the model for several standard benchmarks.

1.1 Related Work

Several different representations for available parallelism have been proposed and studied, such as the component-affinity graph (CAG). Li and Chen were the first to use CAG for modeling the alignment problem[8, 9]. They proposed a heuristic for finding the optimal partitioning of a CAG by selecting one index domain at a time for alignment. Knobe et al. have proposed a framework that is similar to the CAG by Li and Chen [8]. However, unlike our PAL program representation, the CAG does not capture opportunities for dynamic distribution and does not have an underlying performance model.

Kremer et al. [7, 3] modeled the dynamic decomposition problem as an explicit search space that contains all the candidate decompositions for each phase in a program. The optimal dynamic decomposition is formulated as a 0-1 integer linear programming problem. However, the complexity of their optimization algorithms is exponential in the size of the search space. However, algorithms for phase compaction [11] and pruning of search space are developed to

improve the performance of the optimization algorithms. The weakness of this approach is that for large programs, the integer programming problem can take unduly large time to find the solution. In comparison, we use a heuristic for finding the optimal dynamic decomposition.

Sophisticated cost models have been developed by Chatterjee et. al [4], Sussman[12], and Balasundaram et al. [2]. Sussman's cost model works for programs with simple loop-nests. In comparison, we are able to handle codes with complex control structures. Balasundaram et al. introduced the idea of a *training set*, which contain a cost model for computation and communication. The performance estimation algorithm is based on a training-sets model rather than a theoretical model and the success of the method depends to a large extent on the choice of the training set programs used in building the model for performance prediction.

2 Program Representation

In this section, we briefly describe the program representation utilized by our analysis tool. As part of generating the representation, we need to identify the available loop-level parallelism in the program source. Loops are classified as follows:

parallel A loop is considered to be parallel if there are *no loop-carried dependences* in the loop.

pipelined-parallel A loop is considered to be pipelined parallel, if it has one or more loop-carried dependencies in the loop and belongs to a set of nested loops that are fully permutable [13].

serial A loop is considered to be serial if it is neither parallel nor pipelined-parallel according to the criteria stated above.

Our intermediate representation of a program is called the (**P**arallelism **A**nd **L**ocality) (PAL) graph. A program is assumed to consist of a sequence of loop-nests interspersed with control-flow and sequential code. The PAL graph is a control-flow graph of loop-nests in a program. For each loop-nest, there is a corresponding graph node which contains the following information: array references within loop body, data and computation decompositions, and loop performance data. The graph contains other kinds of nodes, namely branch nodes and merge nodes, that

are used to represent control-flow. A computation decomposition is associated with each loop-nest node, and a data decomposition is associated with each data array at each node (i.e., loop nest, branch and merge nodes).Program analysis identifies iterative loops in a program and represents them as control-flow nodes in the PAL graph[2].

When the same data array is accessed in the body of two loop-nests, it is possible for them to have different decomposition at each of the loop-nests. This is captured as a *decomposition-flow* edge in PAL graph. Often there may be no common decomposition for the data array that fully exploits the available parallelism in both the loop nests. In such a case, finding different decompositions for the array in the two loop nests, results in better performance. A decomposition-flow edge represents potential dynamic decomposition for an array. The reorganization of data is done at runtime, hence this cost contributes to the total execution time of the program. Our analysis for computing reaching decompositions[3] is similar to that implemented in the Fortran D compiler[6].

Figure 2 shows the PAL graph for the program in Figure 1. The PAL graph is annotated with performance data obtained by running the program on a uniprocessor system. The nodes in the graph are annotated with the execution time per iteration, and the number of times the corresponding loop nest is visited (trip count) during the execution of the program. Each conditional node is annotated with the number of visits and number of times each branch is taken.

3 Performance Modeling

The scheme for generating message-passing program is closely related to the performance model. The set of data dependencies in a program together with data and computation decompositions determine communication and synchronization in the resulting message-passing program. Our cost model has the following components:

[2] An iterative loop is defined as one whose index variable is neither directly nor indirectly used in array access expressions. Furthermore any enclosing loop of an iterative loop should also be an iterative loop.

[3] If there exists a control-flow path from node A to node B, and the data decomposition of an array at node A is not changed along this path then the data decomposition is said to reach node B.

```
do i = 1, N                          // loop nest L1
do j = 1, M
     A(i,j) = (i + j) / 2
end do
end do
do step = 1, 100                     // iterative loop
     do i = 1, N                     // loop nest L2
     do j = 1, M
          B(i,j) = 0.0
     end do
     end do
     if (step is even) then
          do i = 1, N                // loop nest L3
          do j = 1, M
               B(i,j) = (B(i,j-1) + B(i,j+1) + A(i,j)) / 3
          end do
          end do
     else
          do i = 1, N                // loop nest L4
          do j = 1, M
               B(i,j) = (A(i+1,j) + A(i,j+1)) / 2
          end do
          end do
     end if
     do i = 1, N                     // loop nest L5
     do j = 1, N
          C(i,j) = B(i,j) + ...
     end do
     end do
end do
sum = 0.0
do i = 1, N                          // loop nest L6
do j = 1, N
     sum = sum + A(i,j)
end do
end do
```

Figure 1: A Sample Program

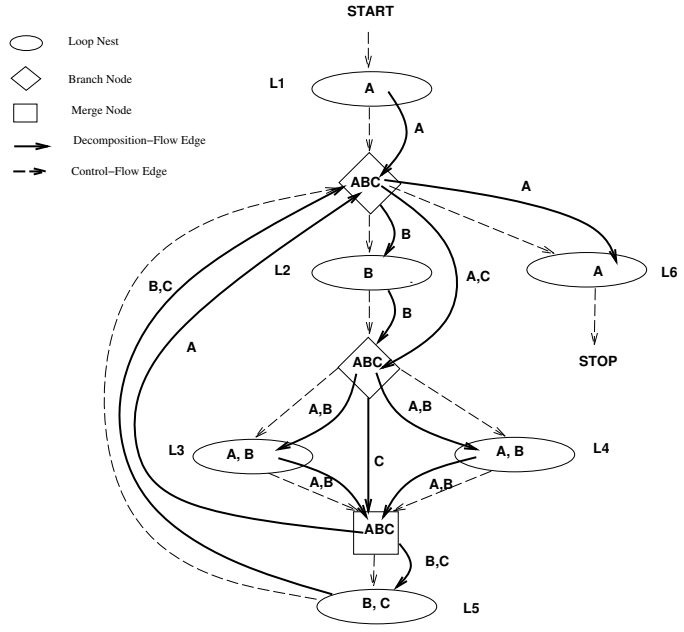

Figure 2: PAL Graph for the Sample Program

1. Cost of a loop nest includes both the cost of computation and the cost of any needed communication. In the message-passing implementation, any accesses to non-local elements of distributed arrays result in messages.

2. Cost of data reorganization during run-time is modeled as a function of array size and machine parameters such as latency, bandwidth, and memory-to-memory copy rate.

3. The contribution of loop nests and data reorganization to the overall completion time of a parallel program is adjusted for the number of times (trips) those operations are performed. Predicted execution time of a parallel program is the total sum of the cost of all loop nests and data reorganizations (if any).

3.1 Cost of Message Communication

Cost of a message represents elapsed time between the start of message send and end of its reception at the remote processor. Interconnect latency is denoted by λ seconds, and throughput is denoted by β bytes per second. Cost of a message of length l bytes is given by:

$$T_{mesg}(l) \;=\; \lambda \;+\; l * \beta \tag{1}$$

3.2 Cost of Loop Nest

Cost of a loop nest consists of (a) computation cost taking into consideration synchronization overhead due to loop-carried dependencies at the distributed loop, and (b) cost of messages for communicating any non-local data. The cost of a loop nest depends on the type of parallelism of the distributed loop, and the array access expressions in the loop body. The method for calculating communication requirements, namely *communication sets*, is identical for loop nests with parallel, pipelined parallel and serial loops.

Cost of Communicating Non-Local Array Elements

Let an array access expression $A(F(\vec{i}))$ occurs in a loop nest $\mathcal{L}$, and D_A and $C_{\mathcal{L}}$ are data and computation decompositions for array A and loop nest $\mathcal{L}$, respectively. The array access

expression refers to a non-local element when the array element $A(F(\vec{i}))$ and the iteration $\vec{i}$ are mapped to different processors. *Communication offset vectors* which are analogous to dependence distance vectors, are useful for capturing non-local accesses in a loop nest. The difference between $D_A(F(\vec{i}))$ and $C_{\mathcal{L}}(\vec{i})$ can be represented as distance vectors, provided $F(\vec{i})$ is affine. These vectors are used in modeling the communication cost in a loop nest.

Size of Non-Local Data

The communication offset specifies the number of rows or columns of an array that needs to be communicated between neighboring processors. This information along with the dimension of the array and how each dimension is distributed is used to calculate the size of the non-local data. The number of elements transported by a unit distance is denoted by the attribute *cssize* whereas the size of an element is denoted by the attribute *elmsize*.

Computation Time

Let t_i be execution time of i^{th} loop, excluding execution time of any inner loops within i^{th} loop. Let T_i be the execution time of i^{th} loop, including execution time for any inner loops. Execution time per iteration of loop j is denoted by W_j. The cost model presented here handles imperfectly-nested loops. Let $(c_1, ..c_n)$ denote the computation decomposition. Say the i^{th} loop is selected for distribution and all other loops are replicated on every processor. i. e. , $c_i = 1$ and $\forall j \neq i . c_j = 0$.

Predicted execution time for the loop nest is:

$$T_{\mathcal{L}} = \sum_{j=1}^{i} X_j ,$$

where

$$
\begin{aligned}
X_j &= T_{parallel}(N, P, W_j, d_<, d_=, d_>), && if\ loop\ j\ is\ parallel\ \&\ c_j = 1 \\
&= T_{pipe}(N, M, P, W_j, d_<, d_=, d_>, b), && if\ loop\ j\ is\ pipelined\ parallel\ \&\ c_j = 1 \\
&= T_{serial}(N, P, W_j, d_<, d_=, d_>), && if\ loop\ j\ is\ serial\ \&\ c_j = 1 \\
&= t_j, && if\ c_j = 0
\end{aligned}
$$

The number of processors is P, parameters N and M describe the size of the iterations space and $d_<, d_=, d_>$ describe the communication sets.

Iterations of a distributed loop are partitioned into blocks of successive iterations, and each block of iterations is assigned to a separate processor. The cost functions $T_{parallel}$, T_{pipe} and T_{serial} are described below.

Parallel Loop

The execution time of a parallel loop is predicted as the sum of cost of executing a block of iterations and the cost of communicating any non-local data (either before or after execution of the loop). For a parallel loop the communication set $d_=$ is empty. The messages corresponding to $d_<$ are performed before the start of the loop whereas the messages corresponding to $d_>$ are performed after the completion of the loop.

$$
\begin{aligned}
T_{parallel}(N, P, c, d_<, d_=, d_>) = \lceil \frac{N}{P} \rceil * c \\
+ \sum_{\forall (A,d) \in (d_< \cup d_= \cup d_>)} T_{mesg}(d * A.cssize * A.elmsize)
\end{aligned}
\tag{2}
$$

Pipelined Parallel Loop

If pipelined-parallel loop is distributed, then it is executed in block-pipelined fashion. This approach is illustrated in Figure 4 with the help of an example that has a 2-D iteration space and a 1-D processor space.

```
do j = 2, M
do i = 2, N
     b(i,j)   = b(i,j)   - 1/b(i-1,j-1)
     u(i,j)   = u(i,j)   + u(i-1,j-1)/b(i,j-1)
end do
end do
```

Figure 3: A Pipeline Parallel Loop Nest

In the figure, each rectangular shaped tile represents a subset of the iterations belonging to the original loop nest. Each processor in the target parallel computer is assigned a row of tiles. The shaded regions in the figure represent the border elements of data arrays that must be communicated to the processor above. The broken line represents a critical path that determines the execution time of the loop nest. The tiles on the critical path must be executed in strict order. In pipelined execution, the results from the previous time step need to be communicated to neighboring processors. So the cost of executing a block of iterations should include the time required to communicate the partial results.

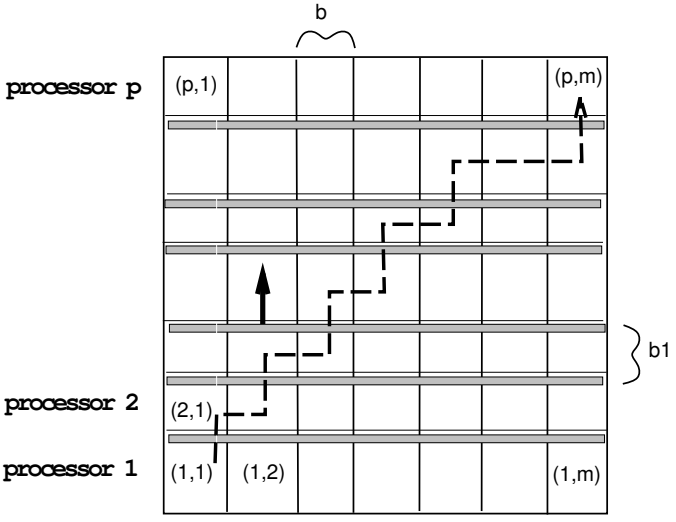

Figure 4: Tiled Iteration Space

The iterations of a distributed loop are evenly divided among the processors for load-balance. Tile size along this dimension is computed as $\lceil \frac{N}{P} \rceil$, where P is the number of processors and N is the total number of iterations. Let us suppose the iteration space is of size $(1:N, 1:M)$. The iterations of the pipelined parallel loop are evenly distributed to the P processors. Let b_1 and b be the tile sizes along the first and second dimensions. Hence $P = \lceil N/b_1 \rceil$. Let $m = \lceil M/b \rceil$. The tile space is of size (P, m). Figure 4 illustrates the mapping of rows of tiles to processors.

Furthermore, the set of data dependencies between iterations is represented by the dependence distance vector $(1,1)$. Therefore tile (t_1, t_2) can not start before tile $(t_1 - 1, t_2 - 1)$ has completed. The tile (t_1, t_2) is executed in time step $t_1 + t_2 - 1$. The cost of the loop nest is the product of number of time-steps and the cost of a single tile.

Tile size along the second dimension, b, is variable. The number of tiles along the second dimension, m, is given by the following equation, where M stands for the number of iterations along the second dimension in the original loop nest:

$$m = \left\lceil \frac{M}{b} \right\rceil$$

When N is not divisible by either b or P, then the tiles belonging to the last column or row are smaller than the others. The cost of block-pipelined execution, T_{pipe} is given by the following formula:

$$\begin{aligned} T_{pipe}(N, N, P, c, d_<, d_=, d_>, b) = & \left(\left\lfloor \frac{N}{b} \right\rfloor + \left\lfloor \frac{N}{\lceil \frac{N}{P} \rceil} \right\rfloor - 1 \right) * \left\lceil \frac{N}{P} \right\rceil * b * c \\ & + max\left((N \bmod b_1) * b,\ b_1 * (N \bmod b)\right) * c \\ & + (N \bmod b_1) * (N \bmod b) * c \\ & + \left(\left\lceil \frac{N}{b} \right\rceil + P - 2 \right) * \sum_{\forall (A,d) \in d_=} T_{mesg}(d * \frac{A.cssize}{N} * b * A.elmsize) \\ & + \sum_{\forall (A,d) \in (d_< \cup d_>)} T_{mesg}(d * A.cssize * A.elmsize) \end{aligned} \tag{3}$$

where, the symbols c and d stand for the computation time of a single iteration and the dependence distance along the first dimension, respectively. Every processor (except the last) must communicate the partial results in d border rows in each tile to the next processor. For

ease of presentation, we assume that $N = M$.

The optimal tile size is obtained by finding the value of b that results in the smallest value for T_{pipe}.

Serial Loop

A serial loop has some loop-carried dependencies that prevent parallel execution of the loop. Partial results computed on a processor are needed at other processor(s). This results in messages being sent and received in the loop nest. The communication corresponding to $d_<$ and $d_>$ can however be performed outside the loop nest. The cost is given by the following formula:

$$T_{serial}(N, P, c, d_<, d_=, d_>) = N * c + \sum_{\forall(A,d) \in (d_< \cup d_>)} T_{mesg}(d * A.cssize * A.elmsize) + (P-1) * \sum_{\forall(A,d) \in d_=} T_{mesg}(d * A.cssize * A.elmsize) \quad (4)$$

3.3 Data Reorganization Cost

Reorganizing a data array during runtime so as to select a different dimension for data distribution involves exchanging data due to change of ownership. We present here a formula for the cost of reorganizing a square matrix of size (N,N). The symbols λ, β and τ represent network latency, bandwidth, and memory-to-memory copy rate.

$$T_{redist}(N, P) = \frac{N}{P} * \left((P-1) * (\lambda + 2 * \frac{\frac{N}{P} * e}{\beta}) + (\lambda + \frac{\frac{N}{P} * e}{\tau}) \right) \quad (5)$$

3.4 Program Execution Time

The overall execution time of a parallel program is estimated as the total sum of execution time for all loop nests plus the cost of any dynamic reorganization of data arrays.

Say $G(V, E)$ is the PAL graph for a program, where V and E denote nodes and control-flow edges. The predicted performance of the program is:

$$cost(G(V,E)) = \sum_{\forall v \in V} LoopNestCost(v) + \sum_{\forall e \in E} RedistributionCost(e) \quad (6)$$

4 Experiments

We provide here an experimental validation of our cost model by reporting accuracy of predicted execution time for several benchmark programs. Furthermore, we show that the model is applicable multiple platforms by reporting results on two target parallel computers, a 16 node IBM SP2, and a Network of Workstations consisting of 12 Sun Ultra workstations connected using Fast Ethernet. Table 1 summarizes parameter of these parallel computers.

Table 1: Cost of Round-Trip Message

platform	OS	CPU	latency (μs)	bandwidth (MB/s)	memory copy (MB/s)
SP2	AIX 4.1.3	SP2 Thin 66MHz	75	32	82
SUN Ultra Cluster	SunOS 5.5	UltraSparc Model 170	320	16	212

4.1 IBM SP2

In this section, we report experiments conducted on IBM SP2 using three benchmark programs (ADI, GRID and Erlebacher) and a synthetic program. The synthetic program is designed to have a wide range of computation to communication ratio that may not be present in the benchmark programs. The candidate computation and data decompositions are suggested by our decomposition tool. Performance of the benchmark programs under the selected decompositions are measured and compared with predicted execution times using our model. Our model is accurate in predicting performance of the benchmark programs within 5% of experimental values.

ADI

This program implements a two-dimensional alternate diagonal implicit (ADI) iterative method which computes the solution for a partial differential equation. It consists of 10 loop nests and has three two-dimensional data arrays of size 512x512. There is an outer iterative loop that is performed 100 times. The program can be divided into two phases; the first phase implements row sweep and the second phase implements a column sweep. Available parallelism is along rows in the first phase and along columns in the second phase. Depending on the number of processors used and the relative speed of interprocessor communication link compared to the processor speed, either a static decomposition with pipelined execution or a dynamic decomposition of the data arrays result in best performance.

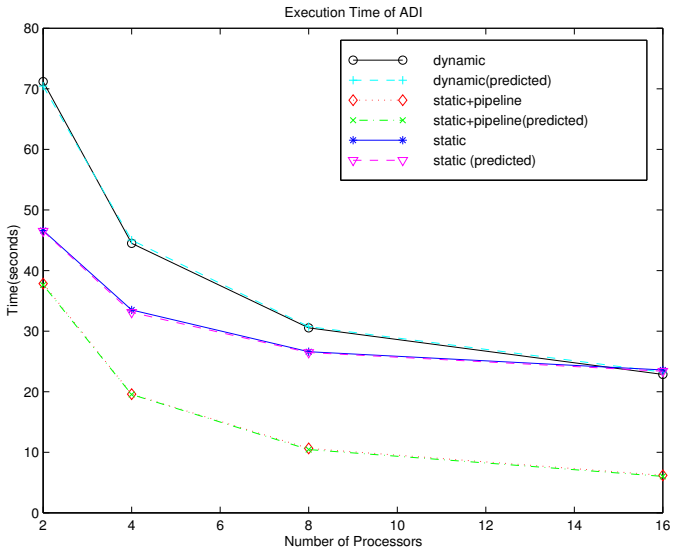

Figure 5: Performance of ADI

We compare performance of the program under three decomposition schemes, namely (a) static decomposition without block-pipelining (b) static data decomposition with block-pipelining, and (c) dynamic decomposition. Estimated and actual execution time corresponding to these decomposition schemes are shown in Figure 5. Our model was accurate in predicting performance and the relative ordering of the decomposition schemes based on estimated and measured per-

formance were the same. The predicted execution times are accurate within $\pm 5\%$ of the actual values.

GRID

The program GRID[4]mplements a successive over-relaxation method for computing the values in a two-dimensional grid using a 9-point stencil. In each iteration, the new values are computed using the values from the previous iteration. As a result, the loops can be executed in parallel. We measured the performance of GRID using static decomposition of data. The overhead due to communication of intermediate results from the previous iteration of the outer iterative loop can be reduced by overlapping the communication with computation. This is similar to block-pipelining case, however the communication patterns differ.

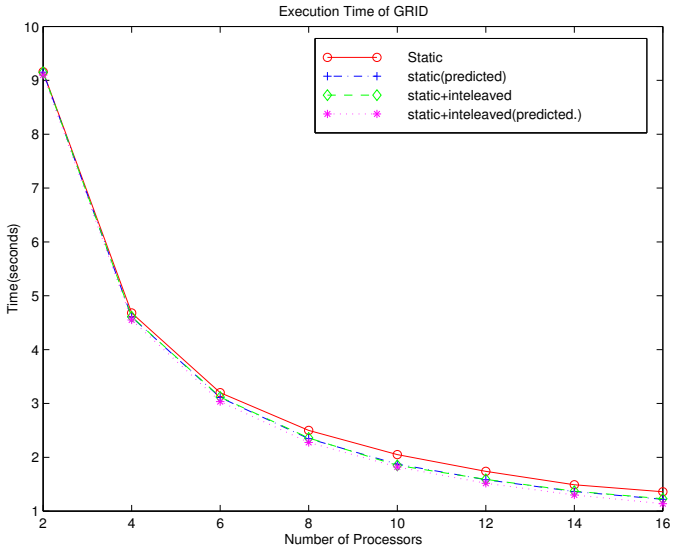

Figure 6: Performance of GRID

For processors sizes 2 to 8, the predicted values (in Figure 6) are within $\pm 5\%$ of the actual execution time of GRID. For processors sizes 10 to 16, the predicted values differed up to

[4] i

10% from the actual values. A likely explanation for this is the contention in the SP2 high-performance switch during collective communication.

Erlebacher

The Erlebacher benchmark program implements a tridiagonal solver for calculating variable derivatives[5]. The program consists of 600 lines of Fortran code and has four three-dimensional data arrays of size 128x128x128. This program consists of three symmetrical phases, one along each of the three dimensions. Any static decomposition that distributes along one of the dimensions of arrays results in effective parallelism in two phases of the program and reduced parallelism in the other phase. However the third phase can be executed in parallel using block-pipelining. It contains no iterative loops or conditional execution of loop nests. Figure 7 shows a plot of execution time for three candidate decomposition schemes. The relative ordering of the decomposition schemes based on the estimated and measured performance were the same. For the dynamic decomposition case, the performance of phase three of the program showed super-linear speedup. This is due to improved locality of memory references.

Synthetic Program

The kernel of the synthetic benchmark is shown in Figure 8. Parallelism is in the first dimension in the first loop nest whereas it is in the second dimension in the second loop nest. The amount of computation in the loop bodies is varied proportional to a workload factor. We conducted five sets of experiments corresponding workload factors (W) 1, 25, 50, 75, and 100. Two candidate decomposition schemes, a static data decomposition with pipelining and a dynamic data decomposition, were considered. The predicted performance is within $\pm 5\%$ of the actual execution time. The results are reported in Tables 2 and 3

Table 2 shows the difference between predicted execution times and actual executions time of the program under static data decomposition with block-pipelining. Table 3 shows the difference of predicted execution times and actual execution times of the program under dynamic data decomposition. The results demonstrate that our cost model is accurate in predicting exe-

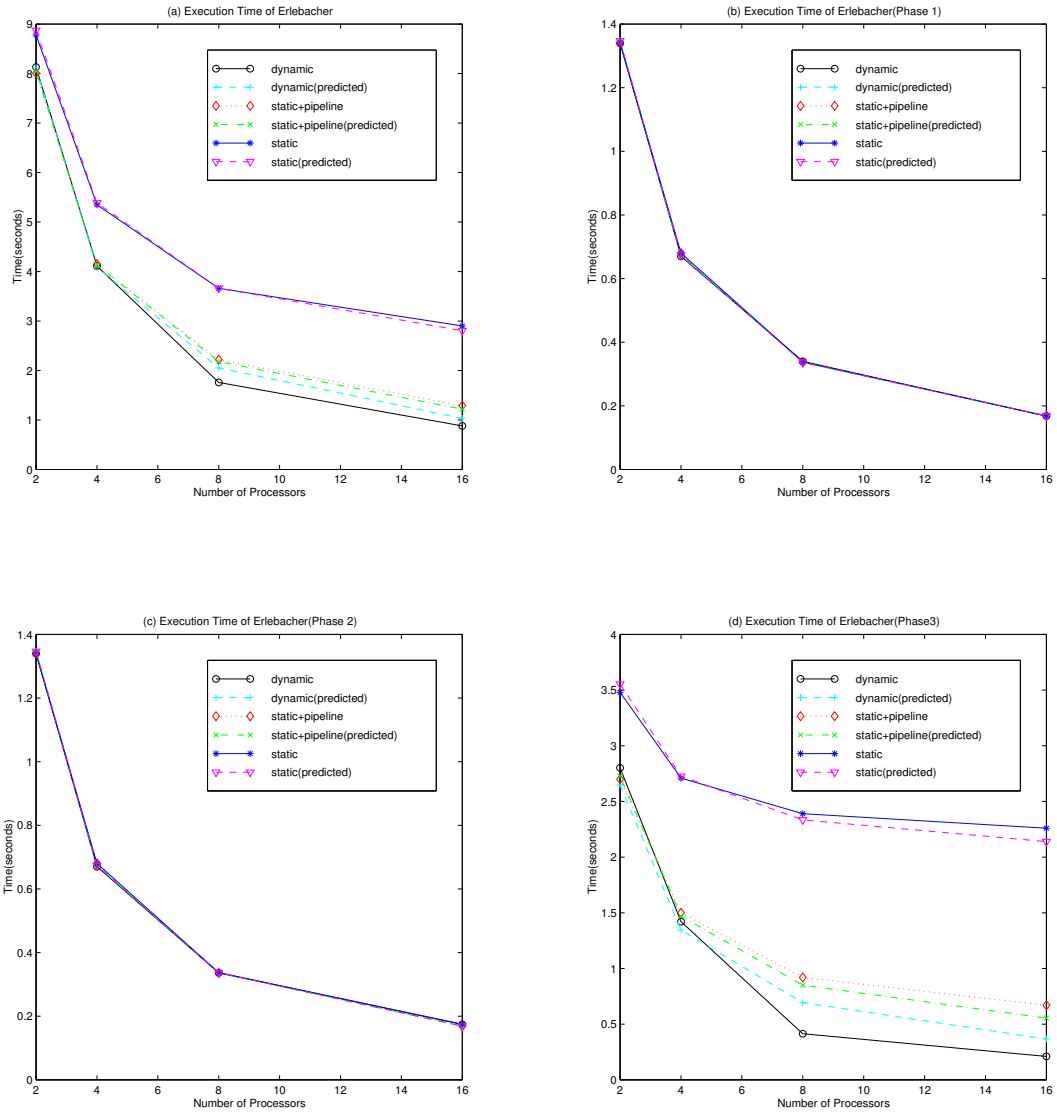

Figure 7: Execution Time of Erlebacher

```
work  = 25
do it = 1, maxitn
  do j = 1, M    // loop nest 1
    do i = 2, N
       do k = 1, work
          A(i,j)  = A(i-1,j)
       end do
    end do
  end do

  do j = 2, N    // loop nest 2
     do i = 1, M
        do k = 1, work
           A(i,j)  = A(i,j-1)
        end do
     end do
  end do
end do
```

Figure 8: Kernel of the Synthetic Program

Table 2: Error in Performance Prediction for Static Decomposition

work	processors			
	2	4	8	16
1	1.7%	-1.5%	-5.0%	-10.7%
25	-5.7%	-3.0%	-4.4%	-6.6%
50	-1.0%	-2.2%	-1.9%	-2.9%
75	0.0%	-0.7%	-1.2%	-2.2%
100	0.3%	-0.4%	-1.3%	-1.9%

Table 3: Error in Performance Prediction for Dynamic Decomposition

work	processors			
	2	4	8	16
1	1.8%	1.6%	0.9%	-0.5%
25	-1.9%	-2.0%	-1.7%	-2.9%
50	-0.3%	-0.8%	-1.4%	-2.1%
75	0.2%	-0.0%	-0.3%	-1.3%
100	0.6%	0.2%	-0.2%	-0.9%

cution time of the benchmark program. The static decomposition scheme with block-pipelining did better than dynamic decomposition scheme for all test cases considered. However if the cost of data reorganization is reduced by interleaving it with computation then the dynamic decomposition scheme is the best.

4.2 Network of Workstations

In this section, we present experimental results obtained on the Network of Workstations (NoW). Predicted and actual execution time of two benchmarks programs (ADI and GRID) are shown in Figure 9. The predicted values are within 6% of the actual values, except in two cases. Estimated cost of reorganizing a data array seem to differ from actual cost for the 8 processor case. In the NoW platform, the overall bandwidth is shared and so any contention for communication link will reduce the effective bandwidth between a pair of nodes.

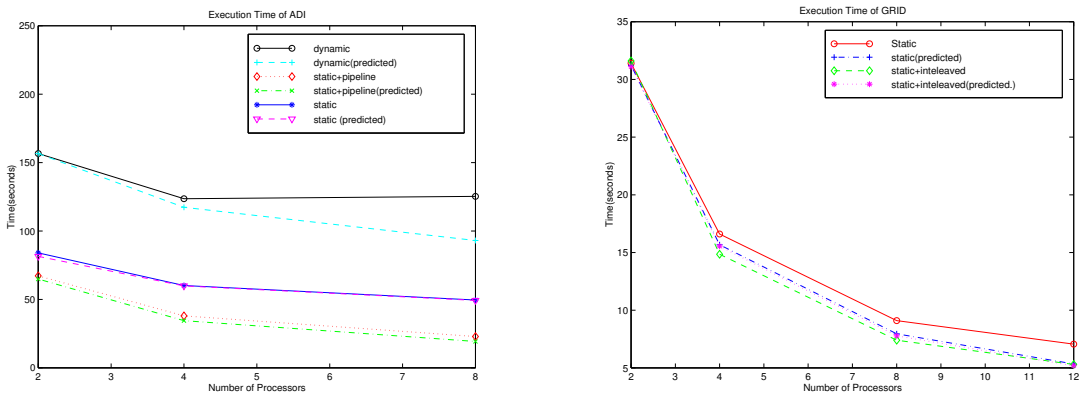

Figure 9: Performance on NoW

5 Conclusion

The model we have developed for predicting the performance of parallel programs is a significant improvement over prior techniques for performance estimation. Our model is simple, based upon an analytical model of communication and computation, and relatively portable

and accurate. The major limitation of the model is its restriction to a message passing style communication and regular dense-matrix style codes. Nevertheless, these restrictions still encompass a wide class of applications and platforms. The restriction to a message passing style communication could be removed by a more elaborate, or platform dependent, cost function for communication. The restriction to regular dense-matrix style codes could be relaxed by incorporating a run-time calculation of data and computation distributions, using an "inspector-executor"[10] model for iterative computations.

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *LNCS*. Springer, August 1993.

[2] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Third ACM SIGPLAN Symposium on Principles and Practce of Parallel Programming*, 1991.

[3] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, November 1993.

[4] Siddhartha Chatterjee, John Gilbert, Robert Schreiber, and Thomas Sheffler. Modeling data-parallel programs with the alignment-distribution graph. *Journal of Parallel and Distributed Computing*, to appear.

[5] T.M. Eidson and G. Erlebacher. Implementation of a fully-balanced periodic tridiagonal solver on a parallel distributed memory architecture. *Concurrency, Practice and Experience*, 7(4):273–302, 1995.

[6] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Support for Scalable Multiprocessors*, New York, 1991. Elsevier.

[7] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Supercomputing*, 1995.

[8] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation (Frontiers'90)*, pages 424–433, October 1990.

[9] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[10] R. Mirchandaney, J.H. Saltz, R.M. Smith, K. Crowley, and D.M. Nicol. Principles of runtime support for parallel processors. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 140–152, July 1988.

[11] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *8th International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *LNCS*, pages 377–391. Springer, August 1995.

[12] Alan Sussman. Model-driven mapping onto distributed memory parallel computers. In *Supercomputing '92*, 1992.

[13] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.